

Komplexität

Ausgangspunkt

- Automatentheorie, die Theorie formaler Sprachen aber auch die Logik, untersuchen die Entscheidbarkeit von Problemen bzw. die Berechenbarkeit von (gewissen) Funktionen.
- Entscheidbarkeit bzw. Berechenbarkeit werden bzgl. formaler Berechnungsmodelle, z.B. dem der Turingmaschine, formal charakterisiert.
- Wenn Probleme entscheidbar bzw. Funktionen berechenbar sind, so bedeutet dies: *prinzipiell entscheidbar* bzw. *prinzipiell berechenbar*. D.h. der Aufwand der Berechnungen kann beliebig gross sein.

Komplexitätstheorie

- Basiskonzepte: *Aufwand*, *Ressource*, *Messen von Komplexität*,...
- Typen: **Zeit**komplexität – **Platz**komplexität
Laufzeit vs. Speicherplatz; time complexity vs. space complexity
- **Komplexitätsklassen**: $P - NP$

→ **Turingmaschinen** (verschiedener Varianten) werden für Komplexitätsuntersuchungen als **Referenzmaschinen** verwendet.

Komplexitätsanalysen: Einführendes Beispiel

Beispiel

- $L = \{a^n b^n \mid n \geq 0\}$ ist eine entscheidbare Sprache.
- Welchen Aufwand benötigt man, um mit einer Turingmaschine das Problem $w \in L?$ zu lösen.
- Das Problem $w \in L?$ kann z.B. durch das folgende Verfahren mit einer Turingmaschine T_1 gelöst werden:
 T_1 : Eingabeband enthält w
 1. Laufe über das Eingabeband und lese die Zeichen: *reject* falls ein a rechts von einem b auftritt.
 2. Falls sowohl a als auch b auf dem Band, wiederhole den Schritt (3):
 3. Laufe über das Eingabeband und streiche ein a und ein b .
 4. Falls alle a ausgestrichen sind, und noch (mind.) ein b auf dem Band steht, oder falls alle b ausgestrichen sind, und noch (mind.) ein a auf dem Band steht, dann *reject*. Anderenfalls, d.h. wenn weder ein a noch ein b übrig sind, dann *accept*.

Aufwandsmaße: worst-case vs. average case

Aufwand

- Messeinheit sind hier Verarbeitungsschritte (als Einheit) \approx Zeitkomplexität
 - D.h. wir unterscheiden auf dieser Ebene nicht den Zeitaufwand verschiedener Basisschritte, z.B. unterscheiden wir in den folgenden Analysen nicht: Lesen, Schreiben, Bewegung des Kopfes um ein Feld.
- Die Laufzeit (gemessen in Schritten) wird als Funktion der Wortlänge betrachtet, z.B. wird der Parameter ‚Grösse des Alphabets‘ nicht berücksichtigt

Worst case analysis: Längste Verarbeitungszeit für eine Eingabe der Länge k .

- z.B. weist T_1 das Wort $ba^n b^n$ schneller zurück, als das Wort $a^n b^{n+1}$
- Was sind die schlimmsten Fälle?

Average case analysis: Durchschnittliche Verarbeitungszeit für Eingaben der Länge k .

- Hier müssen die Klassen von Fällen und deren Verteilung berücksichtigt werden.
- Für die Praxis relevant, aber oft nicht befriedigend zu ermitteln. Ist ein Spezialthema innerhalb der Komplexitätstheorie.

Zeitkomplexität – Laufzeit (bei Turingmaschinen)

Definition 16.1

Sei T eine deterministische Turingmaschine die bei jeder Eingabe anhält, d.h. eine entscheidende Turingmaschine. Die **Laufzeit** oder **Zeitkomplexität** von T ist die Funktion $f: \mathbb{N} \rightarrow \mathbb{N}$, die die **maximale Anzahl von Schritten** angibt, die T für eine Verarbeitung (Entscheidung) eines **Wortes der Länge n** benötigt.

Anmerkungen

- Wenn $f(n)$ die Laufzeit von T ist, so sagen auch, dass
 - T in der Zeit $f(n)$ läuft
 - T eine $f(n)$ -Zeit-Turingmaschine ist.
- Üblicherweise repräsentiert n den Parameter ‚Länge der Zeichenkette‘; die Redeweise wird aber entsprechend auch für andere Parameter verwendet, etwa für ‚Knoten eines Baumes‘ etc, dann aber sollte dies explizit angegeben werden.
- Wir betrachten normalerweise Eingaben einer Länge > 0 und Berechnungen mit mehr als 0 Schritten.

Asymptotische Analyse

- Abschätzung des Zeitaufwands (statt genaue Bestimmung des Zeitaufwands)

Beispiel:

Sei der Aufwand einer Berechnung $f(n) = 6n^3 + 2n^2 + 20n + 45$

- Der Term höchster Ordnung, hier der Term dritter Ordnung, $6n^3$, ist für grosses n „dominant“, mit anderen Worten, der Anteil am Wert von $f(n)$ wird im wesentlichen durch $6n^3$ bestimmt.
- Wir sagen: $f(n)$ ist – abgesehen von einer Konstante c – asymptotisch höchstens so gross wie n^3 .
- Unter Vernachlässigung des Koeffizienten ‚6‘ beschreiben wir das asymptotische Verhalten von f in der folgenden Weise

$f(n)$ ist asymptotisch höchstens n^3 , oder

$f(n)$ ist von der Ordnung n^3

O-Notation

Definition 16.2

Seien f und g Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$. Wir schreiben $f(n) \in O(g(n))$, falls es natürliche Zahlen c und n_0 gibt, so dass für alle $n \geq n_0$ gilt

$$f(n) \leq c g(n)$$

Anmerkungen

- Wenn $f(n) \in O(g(n))$ sagen wir
 - $f(n)$ ist von der Ordnung $g(n)$
 - $g(n)$ ist die (asymptotische) obere Grenze von $f(n)$
- Die Schreibweise $f(n) \in O(g(n))$ ist darin begründet, dass $O(g(n))$ als eine Klasse von Funktionen angesehen werden kann, nämlich der Funktionen, die die (asymptotische) obere Grenze $g(n)$ besitzen.
 - Es gibt auch die Schreibweise $f(n) = O(g(n))$, die wir jedoch nicht verwenden, weil sie eine ‚schludrige‘ (bzw. ‚riskante‘) Verwendung von ‚=‘ beinhaltet.

O-Notation: Beispiele

Beispiel 16.3 $f_1(n) = 6n^3 + 2n^2 + 20n + 45$

- Term höchster Ordnung: $6n^3$
- Vernachlässigung des Koeffizienten ,6': n^3
nach Daumenregel: $f_1(n) \in O(n^3)$
- nach Definition 16.2
 - Setze $c = 7$ und $n_0 = 7$, dann für alle $n \geq n_0$: $6n^3 + 2n^2 + 20n + 45 \leq 7n^3$
 - Also: $f_1(n) \in O(n^3)$

Ausserdem:

$f_1(n) \in O(n^4)$. Wegen $n^4 \geq n^3$, gilt: Wenn $f_1(n) \leq c n^3$, dann $f_1(n) \leq c n^4$. D.h. wenn n^3 asymptotische obere Grenze von $f_1(n)$ ist, dann ist auch n^4 asymptotische obere Grenze von $f_1(n)$.

Aber:

$f_1(n) \notin O(n^2)$. Denn für beliebige c und n_0 gibt es $n \geq n_0$ mit:
 $6n^3 + 2n^2 + 20n + 45 > c n^2$

O-Notation: Logarithmus

- Logarithmus zur Basis 2: $x = \log_2 n \Leftrightarrow 2^x = n$
- Beziehung zwischen Logarithmen zu verschiedenen Basen:
 $\log_b n = \log_2 n / \log_2 b$
Veränderung der Basis bedeutet also die Multiplikation mit einem konstanten Faktor.
- Deswegen ist bei der Schreibweise $f(n) \in O(\log n)$ die Angabe der Basis nicht notwendig.

Beispiel 16.4 $f_2(n) = 3 n \log_2 n + 5 \log_2 \log_2 n + 2$

- Dann $f_2(n) \in O(n \log n)$

O-Notation: „Arithmetik“

- Bei der Komplexitätsanalyse von Algorithmen – bzw. von Turingmaschinen, die Algorithmen realisieren – ist es nützlich Subalgorithmen, bzw. Aktionsfolgen von Turingmaschinen separat zu analysieren.
- Bei der Berechnung der Gesamtkomplexität ergeben sich daher Ausdrücke der Form: $f(n) \in O(n^3) + O(n^2)$.
Definition 16.2 führt (unter Verwendung des Konzeptes) der asymptotischen oberen Grenzen zu $O(n^3) + O(n^2) = O(n^3)$

Typen von asymptotischen oberen Grenzen

- Wenn eine asymptotische obere Grenze von der Form n^c ist, mit $c > 0$, dann heisst sie polynomielle Grenze.
- Wenn eine asymptotische obere Grenze von der Form 2^{n^δ} ist, mit reellem $\delta > 0$, dann heisst sie exponentielle Grenze.

o-Notation

Definition 16.5

Seien f und g Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$. Wir schreiben $f(n) \in o(g(n))$, falls $\lim_{n \rightarrow \infty} f(n) / g(n) = 0$

In anderen Worten: für jede reelle Zahl $c > 0$ existiert ein n_0 , so dass für alle $n \geq n_0$ gilt $f(n) < c g(n)$

Anmerkungen

- Die Beziehung $O(g(n)) \Leftrightarrow o(g(n))$ korrespondiert zur Beziehung $\leq \Leftrightarrow <$
- Beispiele:
 - $\sqrt{n} \in o(n)$
 - $n^2 \in o(n^3)$

Komplexitätsfeinanalyse: Beispiel T_1

- $L = \{a^n b^n \mid n \geq 0\}$

T_1 : Eingabeband enthält w

1. Laufe über das Eingabeband und lese die Zeichen: <i>reject</i> falls ein a rechts von einem b auftritt. [für Rücksetzen zum Bandanfang]	$n + n = 2n$	$O(2n)$
2. Falls sowohl a als auch b auf dem Band, wiederhole den Schritt (3):	$n/2$ Durchgänge vom Typ (3)	$n/2 O(n)$ $= O(n^2)$
3. Laufe über das Eingabeband und streiche ein a und ein b .	n Schritte	n Schritte
4. Falls alle a ausgestrichen sind, und noch (mind.) ein b auf dem Band steht, oder falls alle b ausgestrichen sind, und noch (mind.) ein a auf dem Band steht, dann <i>reject</i> . Anderenfalls, d.h. wenn weder ein a noch ein b übrig sind, dann <i>accept</i> .	n Schritte	$O(n)$
Gesamtaufwand		$O(n^2)$

Komplexitätsklassen

Definition 16.7

Sei $t : \mathbb{N} \rightarrow \mathbb{R}_+$ eine Funktion. Die Zeitkomplexitätsklasse $\text{TIME}(t(n))$ ist die Klasse aller Sprachen die durch eine $O(t(n))$ -Zeit-Turingmaschine entschieden werden kann.

- Die Feinanalyse 15-11 zeigt: $\{a^n b^n \mid n \geq 0\} \in \text{TIME}(n^2)$

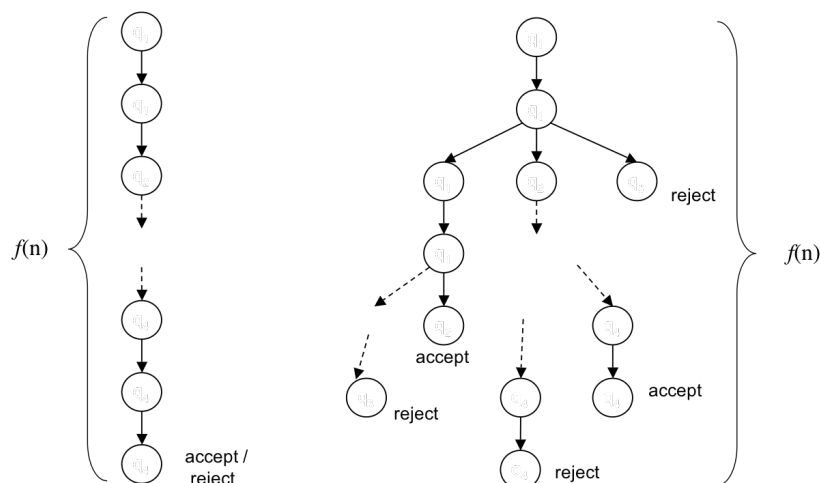
Die nächsten Schritte – die nächsten Fragen:

- Gibt es $\{a^n b^n\}$ entscheidende Turingmaschinen, die geringeren Zeitaufwand haben?
 - Ja: $O(n \log n)$ -Zeit Einband-Turingmaschinen, also $\{a^n b^n \mid n \geq 0\} \in \text{TIME}(n \log n)$
- Was ist der minimale worst-case Zeitaufwand für $\{a^n b^n\}$ Entscheidung mit Einband-Turingmaschinen?
 - $O(n \log n)$. Es gilt: Jede Sprache, die mit $o(n \log n)$ -Zeit Einband-Turingmaschinen entschieden werden kann, ist regulär.
- Ist mit anderen Typen von Turingmaschinen der Zeitaufwand weiter zu verringern?
 - Es gibt $O(n)$ -Zeit Zweiband-Turingmaschinen, die $\{a^n b^n\}$ entscheiden.

Zeitkomplexität und Nichtdeterministische Turingmaschinen

Definition 16.8

Sei NDT eine nichtdeterministische entscheidende Turingmaschine. Die Laufzeit von NDT ist die Funktion $f: \mathbb{N} \rightarrow \mathbb{N}$, die die maximale Anzahl von Schritten (in einem der Zweige des Konfigurationsbaumes) angibt, die NDT für eine Verarbeitung (Entscheidung) eines Wortes der Länge n benötigt.



Die Klasse P

Definition 16.9

P ist die Menge aller Sprachen, die mit polynomiellm Zeitaufwand auf einer deterministischen Ein-Band-Turingmaschine entscheidbar sind.

$$P = \bigcup_{k \in \mathbb{N}} \text{TIME}(n^k)$$

Bemerkung

Die Klasse P ist wichtig, da

- sie auf viele Maschinenmodelle (Varianten von Turingmaschinen, z.B. Mehrbandturingmaschinen) übertragbar ist (invariant bzgl. polynomiell äquivalenten Modellen)
- sie als die Klasse gilt, die alle mit halbwegs realistischem Zeitaufwand lösbaren Probleme enthält.
- Dennoch sind viele hierin enthaltenen Sprachen (Probleme) noch viel zu aufwendig.

Beispiel: Probleme in P

Pfad-Problem

- Gegeben ein gerichteter Graph G und zwei Knoten a, b .
- Gibt es in G einen Pfad von a nach B ?

(Schlechte) Lösung (Verfahren mit exponentiellem Aufwand)

1. Generiere alle Pfade in G . (davon kann es bis zu $2^{|\text{Kanten}|}$ geben)
2. Prüfe die Pfade, ob sie a mit b verbinden.

Polynomielle Lösung: Breitensuch-Verfahren

1. Markiere a .
2. Wiederhole folgendes, bis keine neuen Knoten markiert wurden:
 - 2a. Durchsuche alle Kanten von G .
 - 2ai. Wenn eine Kante (c, d) gefunden wurde, so dass c markiert ist und d nicht markiert ist, dann markiere d .
3. Wenn b markiert ist, dann *akzeptiere*, sonst *weise zurück*.

Aufwand: Anzahl Knoten * Anzahl Kanten

Die Klasse NP

Beispiel

- Ein Hamilton-Pfad in einem Graphen ist ein Pfad, der jeden Knoten genau einmal durchläuft.
- Für das Problem, zu **entscheiden**, ob ein beliebiger gerichteter Graph einen Hamilton-Pfad enthält, hat noch niemand ein Verfahren mit polynomiellm Aufwand gefunden.
- Hat man aber eine Graph und einen Pfad gegeben, kann man mit polynomiellm Aufwand **prüfen**, ob es sich um einen Hamilton-Pfad in dem Graphen handelt.

Definition 16.10

Es sei A eine Sprache. Ein **Prüfer** (**verifier**) für A ist ein Verfahren V , so dass

$$A = \{w \mid V \text{ akzeptiert } wc \text{ für eine Zeichenkette } c\}.$$

Der Zeitaufwand für einen Prüfer wird abhängig von der Länge von w gemessen.

A ist polynomiell **prüfbar** (**verifiable**), wenn A einen Prüfer mit polynomiellm Zeitaufwand hat.

Bemerkung

- Die Arbeit des Prüfers wird durch die Zeichenkette c unterstützt [c kann als zusätzliche Information angesehen werden].
- c wird auch als Zertifikat oder Beweis für die Mitgliedschaft in A bezeichnet.
- Für verschiedene Elemente aus A können für die Prüfung verschiedene Zertifikate erforderlich sein.
- Da der Prüfer (in der Regel) das Zertifikat wenigstens einmal lesen muss, ist bei Prüfern mit polynomiellm Aufwand die Länge des Zertifikats auch durch ein Polynom und die Länge von w beschränkt.

Die Klasse NP

Definition 16.11

NP ist die Menge aller Sprachen, die einen Prüfer mit polynomiellm Zeitaufwand haben.

Theorem 16.12

Eine Sprache A ist genau dann in **NP**, wenn A von einer nicht-deterministischen Turingmaschine mit polynomiellm Zeitaufwand akzeptiert wird.

Beweisidee

Den (deterministischen) Prüfer baut man in eine nicht-deterministische Maschine ein, die zunächst ein Zertifikat nicht-deterministisch erzeugt (rät) und anschließend den Prüfer startet. Da die Zertifikatserzeugung nicht aufwendig ist, ist jede Rechnung insgesamt polynomiell.

Für die andere Richtung muss man im Zertifikat die Entscheidungen der nicht-deterministischen Turingmaschine bei Eingabe von w bei der erfolgreichen Rechnung kodieren. Damit lässt sich ein deterministischer Prüfer konstruieren.

Bemerkung

- Dass $\mathbf{P} \subseteq \mathbf{NP}$, ist offensichtlich.
- Allerdings ist noch nicht bekannt, ob die Inklusion echt ist, oder ob $\mathbf{P} = \mathbf{NP}$.
- Bei der Arbeit an dieser Frage, wurden aber viele wichtige Teilergebnisse erreicht.

Definition 16.13

Eine Funktion $f: \Sigma^* \rightarrow \Sigma^*$ ist genau dann **mit polynomielltem Aufwand berechenbar**, wenn es eine Turingmaschine M mit polynomielltem Zeitaufwand gibt, die f berechnet.

Definition 16.14

- Eine Sprache A über Σ ist genau dann **mit polynomielltem Zeitaufwand auf eine Sprache B über Σ reduzierbar** ($A \leq_p B$), wenn es eine mit polynomielltem Zeitaufwand berechenbare Funktion $f: \Sigma^* \rightarrow \Sigma^*$ gibt, so dass für jedes $w \in \Sigma^*$ gilt:

$$w \in A \Leftrightarrow f(w) \in B.$$

- Die Funktion f wird auch als **polynomielle Zeit-Reduktion** von A auf B bezeichnet.

Reduzierbarkeit als Quasi-Ordnung

Theorem 16.15

Die Relation \leq_p ist reflexiv und transitiv.

Beweis

Reflexivität: Die Funktion Id mit $w = Id(w)$ ist mit konstantem Zeitaufwand berechenbar. Daraus ergibt sich $A \leq_p A$.

Transitivität

Es seien A, B, C Sprachen, so dass $A \leq_p B$ und $B \leq_p C$.

- Damit gibt es Funktionen f_{AB} und f_{BC} , so dass für jedes $w \in \Sigma^*$ gilt:
 $w \in A \Leftrightarrow f_{AB}(w) \in B$ und $w \in B \Leftrightarrow f_{BC}(w) \in C$,
- und Turing-Maschinen $M_{f_{AB}}$ und $M_{f_{BC}}$, die f_{AB} bzw. f_{BC} mit polynomielltem Zeitaufwand berechnen.
- Seien $p_{f_{AB}}$ und $p_{f_{BC}}$ Polynome, die den Zeitaufwand von $M_{f_{AB}}$ und $M_{f_{BC}}$ abschätzen.

Die Idee ist, die Verfahren (bzw. Turing-Maschinen) für die beiden Reduktionsfunktionen so zu kombinieren, dass insgesamt $f_{BC} \circ f_{AB}$ berechnet wird.

Fortsetzung Beweis 16.15

Wir betrachten das folgende Verfahren (bzw. die zugehörige Turing-Maschine genannt $M_{f_{BC} \circ f_{AB}}$):

- Ausgehend von dem Wort w berechnen wir zunächst $f_{AB}(w)$.
- Anschließend verwenden wir $M_{f_{BC}}$, um $f_{BC}(f_{AB}(w))$ zu berechnen.
- Es muss nur noch gezeigt werden, dass die Kombination der Maschinen $f_{BC} \circ f_{AB}$ mit polynomielltem Zeitaufwand berechnet.

Abschätzung des Zeitaufwandes

- Der Zeitbedarf für den ersten Schritt ist ausgehend von w durch $p_{f_{AB}}$ abzuschätzen.
- Der Zeitbedarf für den zweiten Schritt ist ausgehend von $f_{AB}(w)$ durch $p_{f_{BC}}$ abzuschätzen.
- Die Länge von $f_{AB}(w)$ kann nicht größer sein als der Zeitaufwand bei der Berechnung von $f_{AB}(w)$. Also ist $|f_{AB}(w)| \leq c \cdot p_{f_{AB}}(|w|)$.
- Der Zeitbedarf für den zweiten Schritt ist also ausgehend von w durch $p_{f_{BC}} \circ p_{f_{AB}}$ abzuschätzen. ($p_{f_{BC}} \circ p_{f_{AB}}$ ist wieder ein Polynom.)
- Der Zeitaufwand für $M_{f_{BC}} \circ M_{f_{AB}}$ wird insgesamt durch $p_{f_{AB}} + p_{f_{BC}} \circ p_{f_{AB}}$ abgeschätzt.

P und NP plus polynomielle Reduktion

Theorem 16.16

- Wenn eine Sprache A mit polynomielltem Zeitaufwand auf eine Sprache B aus \mathbf{P} (bzw. \mathbf{NP}) reduzierbar ist, dann liegt auch A in \mathbf{P} (bzw. \mathbf{NP}).
- (Man sagt auch: \mathbf{P} und \mathbf{NP} sind abgeschlossen unter polynomieller Reduktion)

Beweis

Die Beweisidee ist genau dieselbe wie bei Theorem 16.15. Nur sind hier die Maschine für die Reduktionsfunktion und die Maschine für die Entscheidungsfunktion hintereinander zu schalten. Die Aufwandsabschätzung ist dann genau so, wie zuvor.

Beobachtung

- Gilt $A \leq_p B$, dann ist (modulo polynomieller Reduktion) B mindestens so aufwendig zu entscheiden, wie A .
[Vorsicht: Reduktionsrichtung \neq Richtung der Aufwandsquasiordnung]
- Sehr einfach zu entscheidende Sprachen kann man immer auch auf kompliziertem Weg bearbeiten, die Umkehrung gilt nicht immer.
- Interessant ist es deshalb auch, obere Schranke und insbesondere die Maxima bzgl. \leq_p in einer Komplexitätsklasse zu kennen.

Definition 16.17

- Eine Sprache B ist genau dann **NP-schwer (NP-hard)**, wenn jede Sprache A aus **NP** mit polynomielltem Zeitaufwand auf B reduzierbar ist.
- (Eine Sprache B ist genau dann **P-schwer (P-hard)**, wenn jede Sprache A aus **P** mit polynomielltem Zeitaufwand auf B reduzierbar ist.)

NP-schwere Sprachen

Definition 16.17 (zur Erinnerung)

- Eine Sprache B ist genau dann **NP-schwer (NP-hard)**, wenn jede Sprache A aus **NP** mit polynomielltem Zeitaufwand auf B reduzierbar ist.

Bemerkung

NP-schwere (bzw. P-schwere) Sprachen bilden bzgl. \leq_p obere Schranken der Klassen **NP** bzw. **P**.

Theorem 16.18

Wenn eine Sprache B NP-schwer ist und sich B mit polynomielltem Zeitaufwand auf die Sprache C reduzieren lässt, dann ist C NP-schwer.

Beweis

Eine einfache Beobachtung zu Ordnungen: Wenn B bzgl. \leq_p eine obere Schranke von **NP** ist und $B \leq_p C$, dann ist auch C bzgl. \leq_p eine obere Schranke von **NP**.

NP-Vollständigkeit

Definition 16.19

- Eine Sprache B ist genau dann **NP-vollständig** (*NP-complete*), wenn sie in **NP** liegt und NP-schwer ist.

Corollar zu 16.18 und 16.19

- Wenn eine Sprache B NP-schwer ist, die Sprache C in **NP** liegt und sich B mit polynomielltem Zeitaufwand auf C reduzieren lässt, dann ist C NP-vollständig.

Beobachtung

- Die NP-vollständigen Sprachen sind die Maxima bzgl. \leq_p in **NP**.
- Die Menge der NP-vollständigen Sprachen bilden eine Äquivalenzklasse bzgl. \leq_p

$P = NP ?$

Theorem 16.20

Wenn es eine Sprache B gibt, die in **P** liegt und NP-schwer ist, dann ist **P = NP**.

Beweis

Ergibt sich direkt aus den Definition 16.17 und Theorem 16.16.

Mitteilung

- Niemand weiß, ob es so eine Sprache B gibt,
- alle (?) vermuten, dass es keine solche Sprache gibt, also dass **P \neq NP**.
- Es gibt (noch ?) keine Beweise (in die eine oder andere Richtung), und einige vermuten, dass es keinen Beweis (in welche Richtung auch immer) geben kann.

- Manchmal findet man Aussagen oder Theoreme des Typs:
,Wenn XYZ, dann **P = NP**’.
Das soll man dann interpretieren als ,Es ist anzunehmen, dass XYZ nicht wahr ist’
(aber wir wissen es noch nicht.)

Definition 16.21

SAT ist die Sprache aller erfüllbaren aussagenlogischen Formeln.

Theorem 16.22

Die Sprache *SAT* ist NP-vollständig.

Beweis

... nicht hier, da (wie zu erwarten ist) viel zu aufwendig für die verfügbare Zeit.

Wichtig zu merken ist aber

- der Inhalt des Theorems
- und, dass das auch bedeutet, dass alle Sprachen / Probleme, in die ein SAT-Problem mit vernünftigem (= polynomiell) Zeitaufwand übersetzt werden kann, NP-schwer und damit nicht effizient entscheidbar sind.

Theorem 16.23

$\mathbf{P} = \mathbf{NP}$ genau dann, wenn $SAT \in \mathbf{P}$.